データ解析およびマシンラーニング向けパフォーマンス・ライブラリー

インテル® DAAL による手書き数字認識向け C++ コードの作成例

Shaojuan Zhu インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

インテル® Data Analytics Acceleration Library (インテル® DAAL) は、インテル® プラットフォーム上でのデータ解析とマシンラーニング向けに最適化されたビルディング・ブロックを提供する、データ解析用のパフォーマンス・ライブラリーです。インテル® DAAL に含まれる関数は、マシンラーニングのすべてのデータ処理段階 (前処理、変換、解析、モデリングから意思決定まで) をカバーします。各処理段階向けに、インテル® Atom™ プロセッサー、インテル® Core™ プロセッサー、インテル® Xeon® プロセッサー、インテル® Xeon Phi™ プロセッサーを含むインテル® アーキテクチャー向けに最適化された関数とユーティリティーが用意されています。インテル® DAAL は、3 つの処理モード (バッチ、オンライン、分散) と 3 つのプログラミング API (C++、Java*、Python*) をサポートします。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

ここでは、インテル® DAAL の手書き数字認識アプリケーションの C++ コード例について見ていきます。手書き数字認識は、典型的なマシンラーニングの問題の 1 つで、いくつかのアプリケーション・アルゴリズムに関連があります。サポート・ベクトル・マシン (SVM)、主成分分析 (PCA)、ナイーブベイズ、ニューラル・ネットワークはすべて、この問題への対応に使用されますが、予測の精度が異なります。インテル® DAAL には、これらのアルゴリズムのほとんどが含まれています。ここでは、SVM を使用して、手書き数字認識にインテル® DAAL のアルゴリズムを導入する方法を示します。

インテル® DAAL でのデータのロード

手書き数字認識において、認識は基本的にマシンラーニング・パイプラインの予測 / 推論段階に当たります。提供された手書き数字から、システムが数字を認識または推論できなければなりません。システムが入力から出力を予測 / 推論できるように、トレーニング・データセットから学習したトレーニング済みモデルが必要になります。トレーニング・モデルを作成する前のステップとして、最初にトレーニング・データを収集します。

サンプル・アプリケーションでは、**UCI Machine Learning Repository** (英語) で公開されている 3,823 件の前処理済みトレーニング・データと 1,797 件の前処理済みテストデータを使用します。インテル® DAAL は、いくつかのデータ形式 (CSV、SQL、HDFS、KDB+) とユーザー定義のデータ形式をサポートしています。ここでは、CSV 形式を使用します。トレーニング・データは `digits_tra.csv` ファイルに、テストデータは `digits_tes.csv` に保存されると仮定します。

インテル® DAAL には、データソースからメモリーへデータをロードするためのいくつかのユーティリティーがあります。ここでは、最初に `trainDataSource` オブジェクトを定義します。これは、CSV ファイルからメモリーへデータをロードすることができる `CSVFeatureManager` です。インテル® DAAL の内部では、メモリーにあるデータは数値テーブルとして保持されます。`CSVFeatureManager` を利用すると、自動的に数値テーブルが作成されます。CSV ファイルからデータをロードするには、メンバー関数 `loadDataBlock()` を呼び出します。これで、データが数値テーブルとしてメモリーにロードされ、後続の処理を行うことができます。CSV ファイルからトレーニング・データをロードする C++ コードの重要な部分を**図 1** に示します。

```
/* 入力データセットのパラメーター */
    string trainDatasetFileName = "digits_tra.csv";
    string testDatasetFileName = "digits_tes.csv";

/* FileDataSource<CSVFeatureManager> を初期化して .csv ファイルから入力データを取得 */
    FileDataSource<CSVFeatureManager> trainDataSource(trainDatasetFileName,
    DataSource::doAllocateNumericTable, DataSource::doDictionaryFromContext);

/* データファイルからデータをロード */
    trainDataSource.loadDataBlock(nTrainObservations);
```

1 トレーニング・データをロードする C++ コード

SVM ベースの手書き数字認識モデルのトレーニング

トレーニング・データがメモリーにロードされたら、そのデータを学習して、トレーニング・モデルを作成します。ここでは、トレーニング・データセットが小さく、データを一度にメモリーに収めることができるため、トレーニング用のアルゴリズムとして、SVM を使用します。処理には、バッチ処理モードを使用します。アルゴリズムを定義後、クラスの数 (この例では 10) などのアルゴリズムに必要な関連パラメーターを設定します。そして、トレーニング・データの数値テーブル `trainDataSource.getNumericTable()` をアルゴリズムに渡します。

`algorithm.compute()` を呼び出すと、SVM の計算が開始され、しばらくするとトレーニングが完了します。トレーニング済みモデルは、`trainingResult` オブジェクトに格納され、`algorithm.getResult()` を呼び出して取得できます。**図 2** にトレーニング・プロセスのサンプルコードを示します。

```
services::SharedPtr<svm::training::Batch<> > training(new
  svm::training::Batch<>());

/* 多クラス SVM トレーニング用のアルゴリズム・オブジェクトを作成 */
    multi_class_classifier::training::Batch<> algorithm;
    algorithm.parameter.nClasses = nClasses;
    algorithm.parameter.training = training;

/* トレーニング・データセットと関連する値をアルゴリズムに渡す */
    algorithm.input.set(classifier::training::data,
    trainDataSource.getNumericTable());

/* 多クラス SVM モデルを作成 */
    algorithm.compute();

/* アルゴリズムの結果を取得 */
    trainingResult = algorithm.getResult();

/* 学習したモデルをディスクファイルへシリアル化 */
    ModelFileWriter writer("./model");
    writer.serializeToFile(trainingResult->get(classifier::training::model));
```

2 トレーニング・プロセスのサンプルコード

インテル® DAAL は、トレーニング済みモデルをメモリーからファイルへ出力するシリアル化関数と、トレーニング済みモデルファイルをメモリーにロードする逆シリアル化関数を提供します。**図 2** の最後の 2 行のように、`model` という名前のファイルに書き込む `ModelFileWriter` を定義します。`writer.serializeToFile()` を呼び出して、`trainingResult` に格納されているトレーニング済みモデルを `model` ファイルに書き込みます。このシリアル化 / 逆シリアル化ユーティリティーは、トレーニング後、サーバーがトレーニング済みモデルをクライアントへ移植し、クライアントがトレーニングを行わずにそれを使用して予測 / 推論を行う場合に役立ちます。`model` ファイルの利用法については、「手書き数字認識アプリケーション」セクションで説明します。

トレーニング済みモデルのテスト

トレーニング済みモデルを使用してテストを行うことができます。トレーニング・プロセスと同様に、**UCI Machine Learning Repository** (英語) からのテストデータを CSV ファイルに保存し、testDataSource.loadDataBlock() を利用してロードします。予測用の SVM アルゴリズム・オブジェクト algorithm を定義する必要があります。algorithm には、テストデータとトレーニング済みモデルの 2 つの入力があります。テストデータは、testDataSource.getNumericTable() で渡します。バッチテストでは、トレーニング済みモデルは trainingResult->get() で渡します。(トレーニング済みモデルをファイルで渡す方法は、「手書き数字認識アプリケーション」セクションで示します。)

アルゴリズムの入力の設定が完了したら、algorithm.compute() を呼び出してテストプロセスを完了します。**図 3** は、テストプロセスのサンプルコードです。テスト後の予測結果は、algorithm.getResult() を呼び出して取得します。

```
services::SharedPtr<svm::prediction::Batch<> > prediction(new
  svm::prediction::Batch<>());

/* FileDataSource<CSVFeatureManager> を初期化して .csv ファイルからテストデータを取得 */
       FileDataSource<CSVFeatureManager> testDataSource(testDatasetFileName,
              DataSource::doAllocateNumericTable,
              DataSource::doDictionaryFromContext);
       testDataSource.loadDataBlock(nTestObservations);

/* 多クラス SVM 値の予測用のアルゴリズム・オブジェクトを作成 */
       multi_class_classifier::prediction::Batch<> algorithm;

       algorithm.parameter.prediction = prediction;

/* テスト・データセットとトレーニング済みモデルをアルゴリズムに渡す */
       algorithm.input.set(classifier::prediction::data,
              testDataSource.getNumericTable());
       algorithm.input.set(classifier::prediction::model,
              trainingResult->get(classifier::training::model));

/* 多クラス SVM 値を予測 */
       algorithm.compute();

/* アルゴリズムの結果を取得 */
       predictionResult = algorithm.getResult();
```

3 テストプロセスのサンプルコード

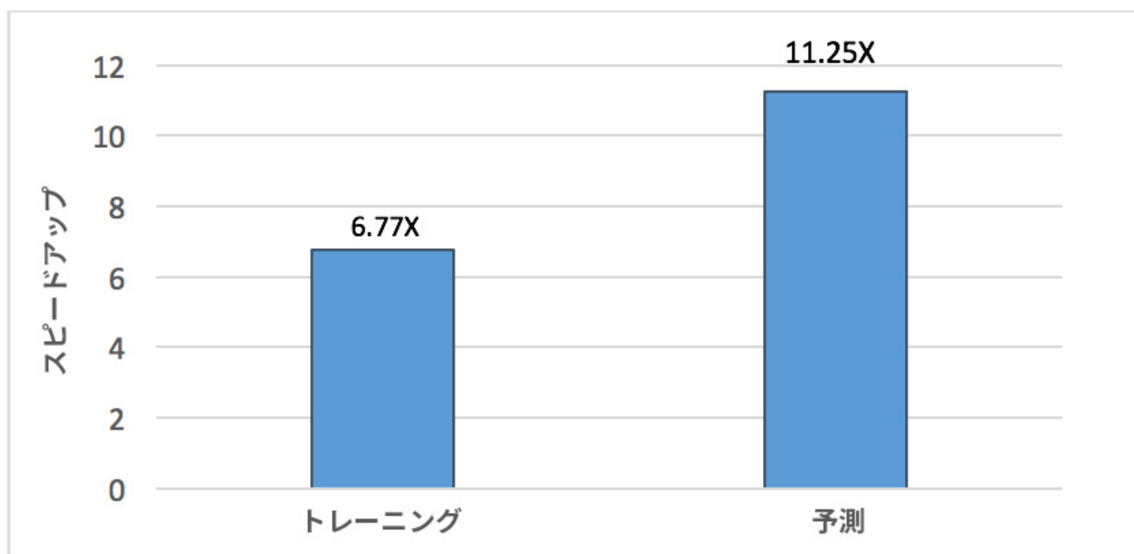
インテル® DAAL には、混同行列、平均正解率、不正解率などの品質メトリックを計算する関数も含まれています。**図 4** は、**UCI Machine Learning Repository** (英語) からのテスト・データセットに対し SVM を利用した場合の品質メトリックです。テストデータ全体の平均正解率が 99.6% であることが分かります。

```
Confusion matrix:
177.000  0.000    0.000    0.000    1.000    0.000    0.000    0.000    0.000    0.000
0.000    181.000  0.000    0.000    0.000    0.000    0.000    0.000    1.000    0.000
0.000    2.000    173.000  0.000    0.000    0.000    0.000    1.000    1.000    0.000
0.000    0.000    0.000    176.000  0.000    1.000    0.000    0.000    3.000    3.000
0.000    1.000    0.000    0.000    179.000  0.000    0.000    0.000    1.000    0.000
0.000    0.000    0.000    0.000    0.000    180.000  0.000    0.000    0.000    2.000
0.000    0.000    0.000    0.000    0.000    0.000    180.000  0.000    1.000    0.000
0.000    0.000    0.000    0.000    0.000    0.000    0.000    170.000  1.000    8.000
0.000    3.000    0.000    0.000    0.000    0.000    0.000    0.000    166.000  5.000
0.000    0.000    0.000    2.000    0.000    1.000    0.000    0.000    2.000    175.000

Average accuracy: 0.996
Error rate:      0.004
Micro precision: 0.978
Micro recall:    0.978
Micro F-score:   0.978
Macro precision: 0.978
Macro recall:    0.978
Macro F-score:   0.978
```

4 品質メトリック

インテル® DAAL と scikit-learn の SVM パフォーマンスの比較

SVM アルゴリズムは、多くのマシンラーニング・フレームワークやライブラリーのパッケージで採用されている古典的なアルゴリズムです。scikit-learn は、よく用いられるマシンラーニング向けの Python* ライブラリーです。ここでは、scikit-learn の SVM 分類を利用する手書き数字アプリケーションで同じトレーニングとトレーニング・データを使用し、インテル® DAAL と scikit-learn の SVM パフォーマンス (トレーニングと予測) を比較しました。結果は、**図 5** に示すとおりです。



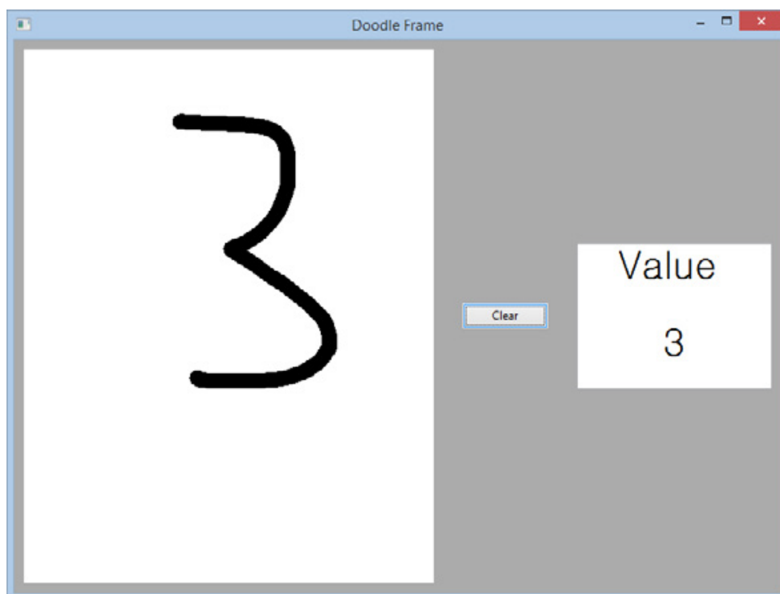
5 インテル® プロセッサー上でのインテル® DAAL と scikit-learn の SVM パフォーマンスの向上

テストに使用したシステムは、インテル® Xeon® プロセッサー E5-2680 V3 @ 2.50GHz、24 コア、CPU ごとに 30MB L3 キャッシュ、256GB RAM です。オペレーティング・システムは、Red Hat* Enterprise Linux* Server 6.6 (64 ビット) です。各ライブラリーのバージョンは、インテル® DAAL 2016 Update 2 と scikit-learn 0.16.1 です。

図 5 に示すように、トレーニングとテストの両方において、インテル® DAAL の SVM パフォーマンスのほうが scikit-learn よりも優れています。インテル® DAAL の SVM は scikit-learn よりも、トレーニングでは 6.77 倍、テスト / 予測では 11.25 倍高速です。

手書き数字認識アプリケーション

前述のように、サーバー側で学習したトレーニング済みモデルを、クライアントに移植して使用できます。ここでは、トレーニング済みモデルを移植可能な、単純な手書き数字認識アプリケーションを使用します。このインタラクティブなアプリケーションには、予測 / テスト段階しかありません。**図 6** は、アプリケーションのインターフェイスです。2 つの白いパネルボックスがあります。左の大きな白いパネルボックスに、0 から 9 の数字を 1 つ書き込みます。右の小さな白いパネルボックスは、システムが認識した左のパネルボックスに書き込まれた数字を表示します。**図 6** では、左のパネルボックスに 3 と書き込まれており、システムはこれを認識して、推論結果 3 を右のパネルボックスに表示しています。



6 数字認識アプリケーション

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

このアプリケーションでは、一度に 1 つの数字のみテスト / 推論します。書き込まれた数字に対し、いくつかの前処理手法により手書き数字から抽出された特徴を含むテスト CSV ファイルが生成されます。(前処理は、この記事の趣旨から外れるため、ここでは取り上げません。) テストデータを取得できたので、`algorithm` オブジェクトのもう 1 つの入力であるトレーニング済みモデルについて見ていきましょう。

前述のように、トレーニング済みモデルは作成済みで、`model` ファイルにあります。ここでは、トレーニング済みモデルをファイルからロードします (つまり、モデルをメモリーへ逆シリアル化します)。`ModelFileReader` を定義して、`reader.deserializeFromFile(pModel)` を呼び出し、`model` ファイルから読み取ります。`pModel` はモデルへのポインターです。**図 7** に C++ コードを示します。大部分のコードは、**図 3** と同じです。`algorithm.compute()` が完了したら、入力された数字に対するラベル / 予測された数字を含む予測結果 `predictionResult1` を取得します。

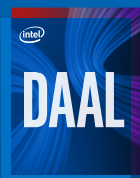
```
services::SharedPtr<svm::prediction::Batch<> > prediction1(new
   svm::prediction::Batch<>());

/* FileDataSource<CSVFeatureManager> を初期化して .csv ファイルからテストデータを取得 */
        FileDataSource<CSVFeatureManager> testDataSource(testDatasetFileName,
                DataSource::doAllocateNumericTable,
                DataSource::doDictionaryFromContext);
        testDataSource.loadDataBlock(1);

/* 多クラス SVM 値の予測用のアルゴリズム・オブジェクトを作成 */
        multi_class_classifier::prediction::Batch<> algorithm;

        algorithm.parameter.prediction = prediction1;

/* ディスクファイルからモデルを逆シリアル化 */
        ModelFileReader reader("./model");
        services::SharedPtr<multi_class_classifier::Model> pModel(new
multi_class_classifier::Model());
        reader.deserializeFromFile(pModel);

/* テスト・データセットとトレーニング済みモデルをアルゴリズムに渡す */
        algorithm.input.set(classifier::prediction::data,
testDataSource.getNumericTable());
        algorithm.input.set(classifier::prediction::model, pModel);

/* 多クラス SVM 値を予測 */
        algorithm.compute();

/* アルゴリズムの結果を取得 */
        predictionResult1 = algorithm.getResult();

/* 予測されたラベルを取得 */
        predictedLabels1 = predictionResult1->get(classifier::prediction::prediction);
```

7 C++ コード

まとめ

インテル® DAAL は、マシンラーニングのパイプライン全体に対応したビルディング・ブロックを提供します。ここでは、SVM の C++ コード例を用いて、アプリケーションでインテル® DAAL を使用して、ファイルからデータをロードし、トレーニング・モデルを作成し、トレーニング済みモデルを適用する方法を紹介しました。

インテル® DAAL の関数はインテル® アーキテクチャー向けに最適化されているため、マシンラーニング・アプリケーションにインテル® DAAL のビルディング・ブロックを利用することで、インテル® プラットフォーム上で最良のパフォーマンスが得られます。ここで紹介したように、インテル® DAAL の SVM は、scikit-learn の SVM よりもパフォーマンスが優れています。



インテル® DAAL の詳細 >